



RNNs and Transformers

Jim Dowling
jdowling@kth.se
2022-11-23

Slides by Francisco J. Pena, Amir H. Payberah, and Jim Dowling



Let's Start With An Example



Google

the students opened their



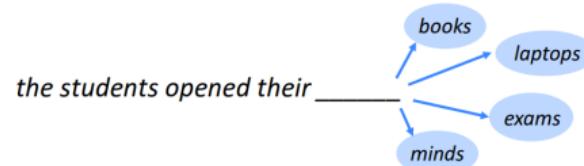
- their work
- their books
- their teachers
- their homework
- their lecturer
- their new lecturer

Feeling Lucky

venska

Language Modeling (1/2)

- ▶ Language modeling is the task of predicting what word comes next.

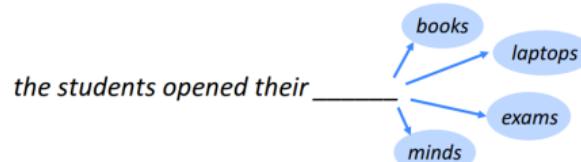


Language Modeling (2/2)

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$p(x^{(t+1)} = w_j | x^{(t)}, \dots, x^{(1)})$$

- w_j is a word in vocabulary $V = \{w_1, \dots, w_v\}$.





n-gram Language Models

- ▶ the students opened their ...
- ▶ How to learn a Language Model?
- ▶ Learn a n-gram Language Model!
- ▶ A n-gram is a chunk of n consecutive words.
 - Unigrams: "the", "students", "opened", "their"
 - Bigrams: "the students", "students opened", "opened their"
 - Trigrams: "the students opened", "students opened their"
 - 4-grams: "the students opened their"
- ▶ Collect statistics about how frequent different n-grams are, and use these to predict next word.



n-gram Language Models - Example

- ▶ Suppose we are learning a **4-gram Language Model**.

- $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the students opened their~~ _____
discard _____
condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ In the corpus:

- "students opened their" occurred 1000 times
 - "students opened their books" occurred 400 times:
 $p(\text{books} | \text{students opened their}) = 0.4$
 - "students opened their exams" occurred 100 times:
 $p(\text{exams} | \text{students opened their}) = 0.1$



Problems with n-gram Language Models - Sparsity

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ What if "`students opened their` w_j " never occurred in data? Then w_j has probability 0!
- ▶ What if "`students opened their`" never occurred in data? Then we can't calculate probability for any w_j !
- ▶ Increasing n makes **sparsity** problems worse.
 - Typically we can't have n bigger than 5.



Problems with n-gram Language Models - Storage

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- ▶ For "students opened their w_j ", we need to store count for all possible 4-grams.
- ▶ The model size is in the order of $O(\exp(n))$.
- ▶ Increasing n makes model size huge.

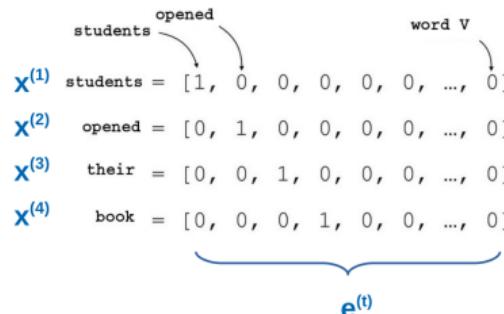
Can We Build a Neural Language Model? (1/3)

► Recall the **Language Modeling** task:

- **Input:** sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
- **Output:** probability dist of the next word $p(x^{(t+1)} = w_j | x^{(t)}, \dots, x^{(1)})$

► One-Hot encoding

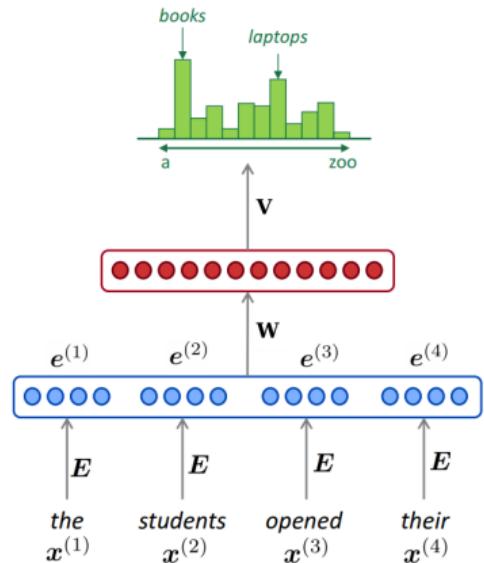
- Represent a **categorical variable** as a **binary vector**.
- All recodes are **zero**, except the index of the integer, which is **one**.
- Each embedded word $e^{(t)} = E^T x^{(t)}$ is a **one-hot vector** of size **vocabulary size**.



Can We Build a Neural Language Model? (2/3)

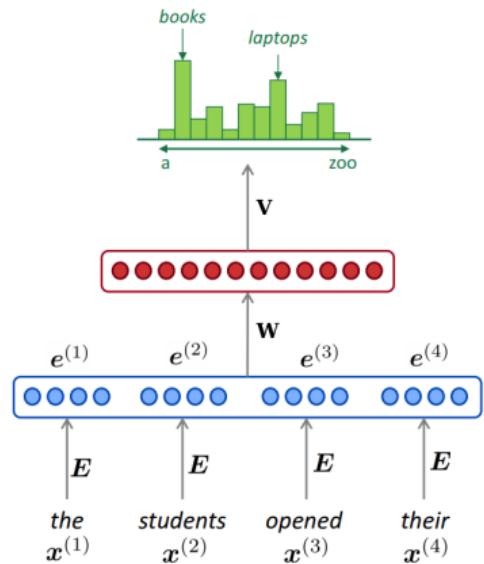
► A MLP model

- Input: words $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$
- Input layer: one-hot vectors $e^{(1)}, e^{(2)}, e^{(3)}, e^{(4)}$
- Hidden layer: $\mathbf{h} = f(\mathbf{w}^\top \mathbf{e})$, f is an activation function.
- Output: $\hat{\mathbf{y}} = \text{softmax}(\mathbf{v}^\top \mathbf{h})$



Can We Build a Neural Language Model? (3/3)

- ▶ **Improvements** over n-gram LM:
 - No sparsity problem
 - Model size is $O(n)$ not $O(\exp(n))$
- ▶ Remaining problems:
 - It is **fixed 4** in our example, which is small
 - We need a neural architecture that can process any length input





Recurrent Neural Networks (RNN)

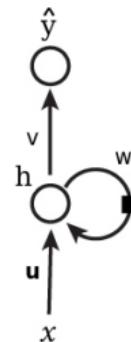


Recurrent Neural Networks (1/4)

- ▶ The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
 - Until here, we assume that all inputs (and outputs) are independent of each other.
 - Independent input (output) is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).
- ▶ They can analyze time series data and predict the future.
- ▶ They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs.

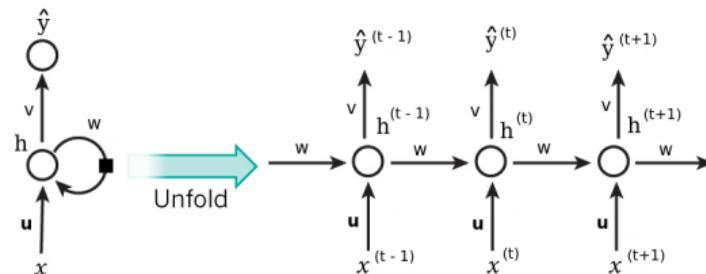
Recurrent Neural Networks (2/4)

- ▶ Neurons in an **RNN** have **connections pointing backward**.
- ▶ RNNs have **memory**, which captures **information** about what **has been calculated so far**.



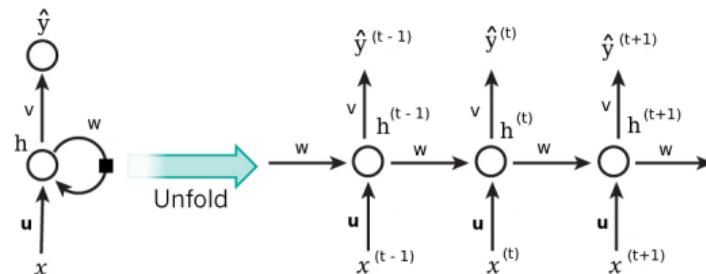
Recurrent Neural Networks (3/4)

- ▶ **Unfolding the network:** represent a network against the time axis.
 - We write out the network for the **complete sequence**.
- ▶ For example, if the sequence we care about is a **sentence of three words**, the network would be **unfolded** into a **3-layer** neural network.
 - One layer for each word.



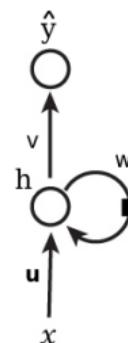
Recurrent Neural Networks (4/4)

- ▶ $h^{(t)} = f(u^T x^{(t)} + wh^{(t-1)})$, where f is an activation function, e.g., `tanh` or `ReLU`.
- ▶ $\hat{y}^{(t)} = g(vh^{(t)})$, where g can be the `softmax` function.
- ▶ $\text{cost}(y^{(t)}, \hat{y}^{(t)}) = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = -\sum y^{(t)} \log \hat{y}^{(t)}$
- ▶ $y^{(t)}$ is the `correct` word at time step t , and $\hat{y}^{(t)}$ is the `prediction`.



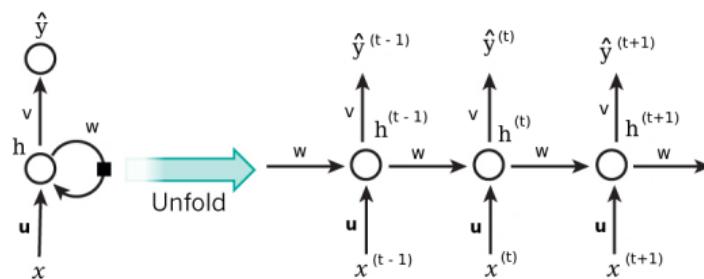
Recurrent Neurons - Weights (1/4)

- ▶ Each recurrent neuron has **three sets of weights**: **u**, **w**, and **v**.



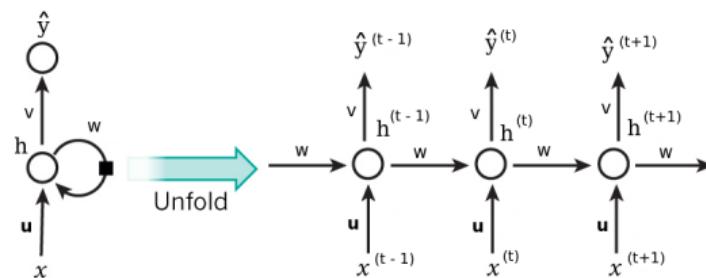
Recurrent Neurons - Weights (2/4)

- ▶ **u**: the **weights for the inputs $x^{(t)}$** .
- ▶ **$x^{(t)}$** : is the **input at time step t**.
- ▶ For example, **$x^{(1)}$** could be a **one-hot vector** corresponding to the **first word of a sentence**.



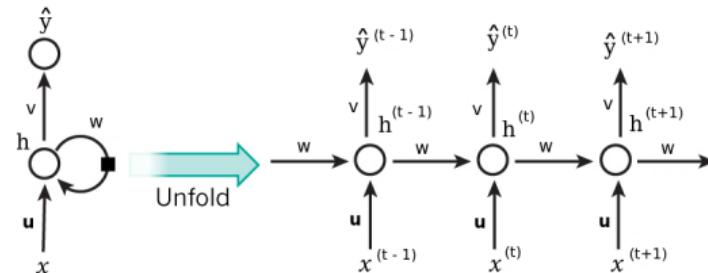
Recurrent Neurons - Weights (3/4)

- ▶ w : the **weights** for the **hidden state** of the **previous time step** $h^{(t-1)}$.
- ▶ $h^{(t)}$: is the **hidden state (memory)** at time step t .
 - $h^{(t)} = \tanh(u^T x^{(t)} + wh^{(t-1)})$
 - $h^{(0)}$ is the **initial hidden state**.



Recurrent Neurons - Weights (4/4)

- ▶ v : the **weights** for the **hidden state** of the **current time step** $h^{(t)}$.
- ▶ $\hat{y}^{(t)}$ is the **output** at step t .
- ▶ $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$
- ▶ For example, if we wanted to **predict the next word** in a sentence, it would be a **vector of probabilities** across our vocabulary.

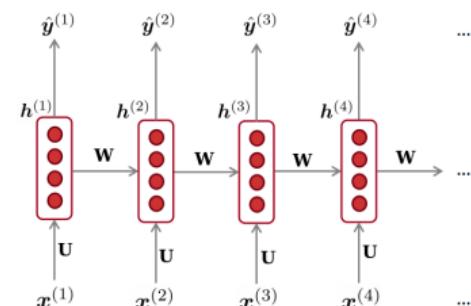
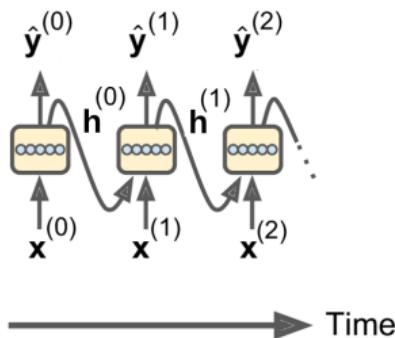


Layers of Recurrent Neurons

- At each time step t , every neuron of a layer receives both the input vector $\mathbf{x}^{(t)}$ and the output vector from the previous time step $\mathbf{h}^{(t-1)}$.

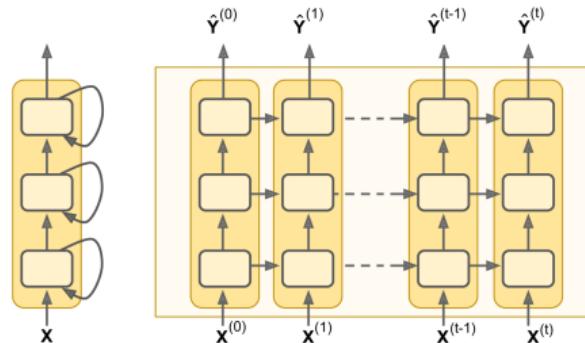
$$\mathbf{h}^{(t)} = \tanh(\mathbf{u}^T \mathbf{x}^{(t)} + \mathbf{w}^T \mathbf{h}^{(t-1)})$$

$$\mathbf{y}^{(t)} = \text{sigmoid}(\mathbf{v}^T \mathbf{h}^{(t)})$$



Deep RNN

- ▶ Stacking **multiple layers** of cells gives you a **deep RNN**.

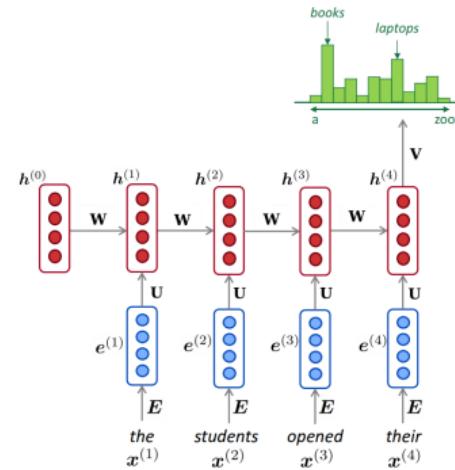
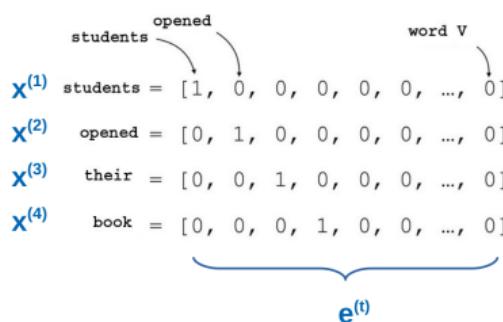




Let's Back to Language Model Example

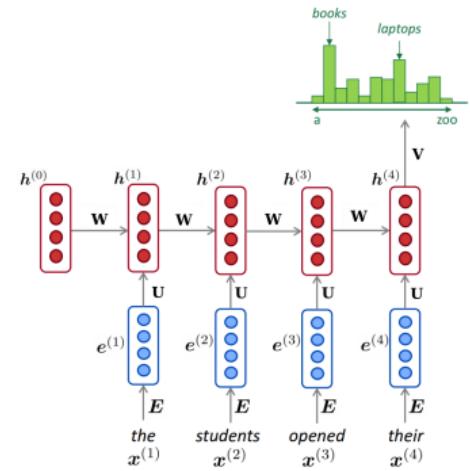
A RNN Neural Language Model (1/2)

- ▶ The input x will be a **sequence of words** (each $x^{(t)}$ is a **single word**).
- ▶ Each embedded word $e^{(t)} = E^T x^{(t)}$ is a **one-hot vector** of size **vocabulary size**.



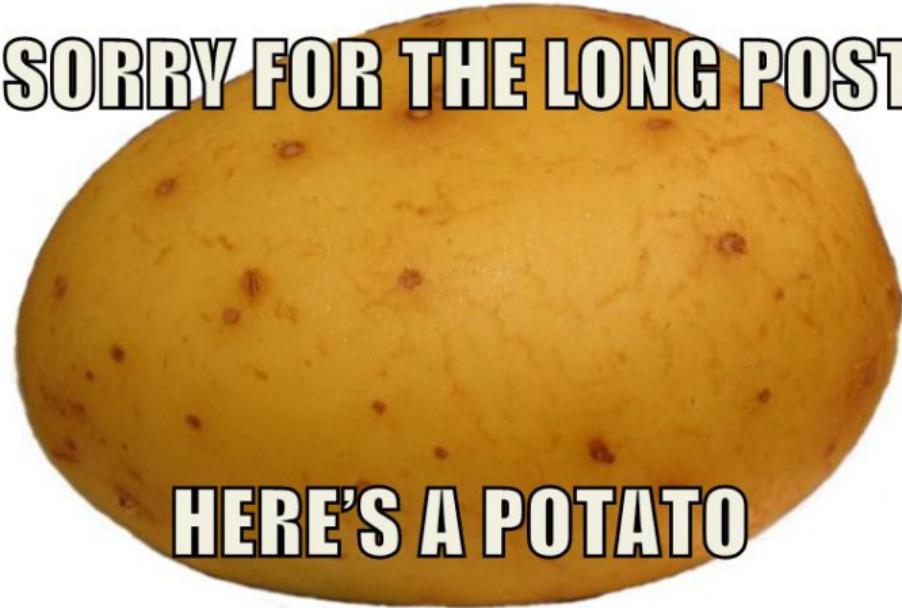
A RNN Neural Language Model (2/2)

- ▶ Let's recap the equations for the RNN:
 - $h^{(t)} = \tanh(\mathbf{u}^\top \mathbf{e}^{(t)} + \mathbf{w}h^{(t-1)})$
 - $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{v}h^{(t)})$
- ▶ The output $\hat{\mathbf{y}}^{(t)}$ is a vector of **vocabulary size** elements.
- ▶ Each element of $\hat{\mathbf{y}}^{(t)}$ represents the **probability** of that word being the **next word** in the sentence.





SORRY FOR THE LONG POST



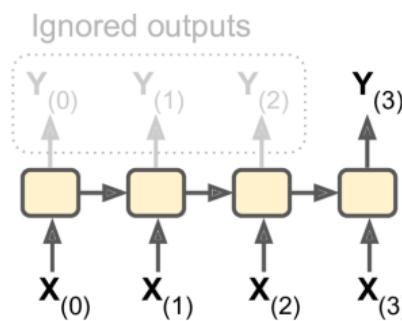
HERE'S A POTATO



RNN Design Patterns

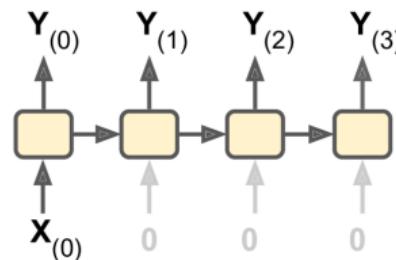
RNN Design Patterns - Sequence-to-Vector

- ▶ **Sequence-to-vector** network: takes a **sequence of inputs**, and ignore all outputs except for the last one.
- ▶ E.g., you could feed the network a **sequence of words** corresponding to a movie review, and the network would output a **sentiment score**.



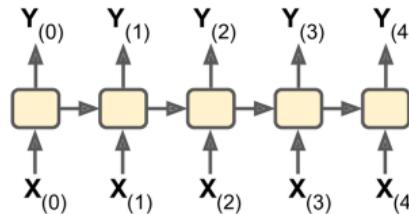
RNN Design Patterns - Vector-to-Sequence

- ▶ **Vector-to-sequence** network: takes a **single input** at the first time step, and let it **output a sequence**.
- ▶ E.g., the input could be an **image**, and the output could be a **caption** for that image.



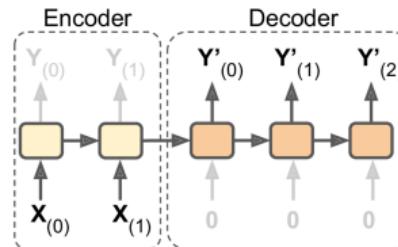
RNN Design Patterns - Sequence-to-Sequence

- ▶ **Sequence-to-sequence** network: takes a **sequence of inputs** and produce a **sequence of outputs**.
- ▶ Useful for **predicting time series such as stock prices**: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future.
- ▶ Here, both input sequences and output sequences have the **same length**.



RNN Design Patterns - Encoder-Decoder

- ▶ **Encoder-decoder** network: a **sequence-to-vector** network (**encoder**), followed by a **vector-to-sequence** network (**decoder**).
- ▶ E.g., **translating** a sentence from one language to another.
- ▶ You would feed the network **a sentence in one language**, the encoder would convert this sentence into a **single vector representation**, and then the decoder would decode this vector into a sentence in another language.

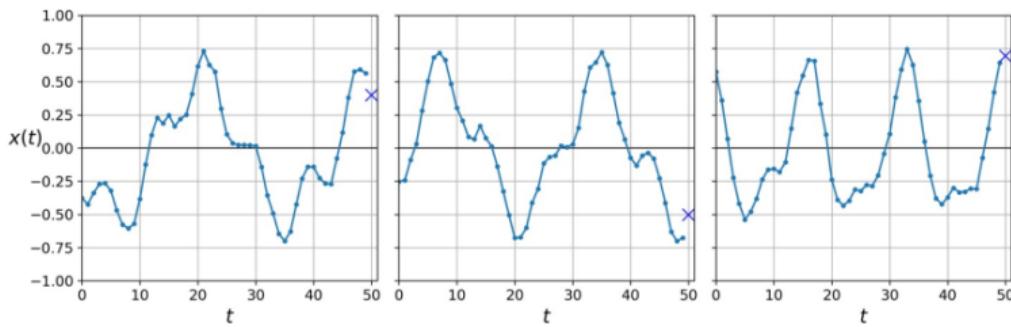




RNN in TensorFlow

RNN in TensorFlow (1/5)

- ▶ Forecasting a **time series**
- ▶ E.g., a dataset of 10000 time series, each of them **50 time steps long**.
- ▶ The goal here is to **forecast the value at the next time step** (represented by the X) for each of them.





RNN in TensorFlow (2/5)

- ▶ Use fully connected network

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.003993967570985357
```



RNN in TensorFlow (3/5)

► Simple RNN

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

model.compile(loss="mse", optimizer='adam')
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.011026302369932333
```



RNN in TensorFlow (4/5)

► Deep RNN

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.003197280486735205
```



RNN in TensorFlow (5/5)

- ▶ Deep RNN (second implementation)
- ▶ Make the second layer return only the last output (no `return_sequences`)

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20)

model.evaluate(X_test, y_test, verbose=0)
# loss: 0.002757748544837038
```



Training RNNs

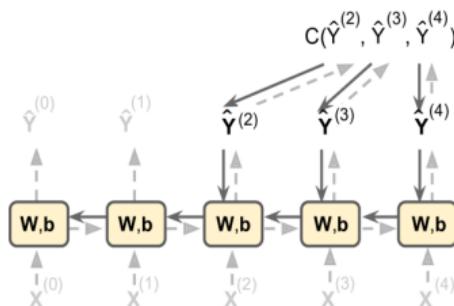


Training RNNs

- ▶ To train an RNN, we should unroll it through time and then simply use regular backpropagation.
- ▶ This strategy is called backpropagation through time (BPTT).

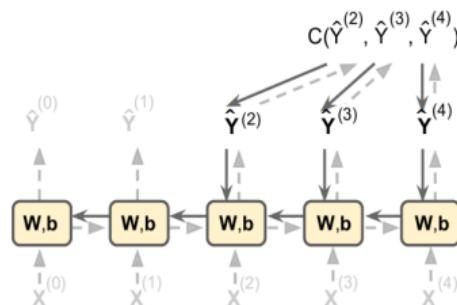
Backpropagation Through Time (1/3)

- ▶ To train the model using **BPTT**, we go through the following steps:
- ▶ 1. **Forward pass** through the **unrolled network** (represented by the dashed arrows).
- ▶ 2. The **cost function** is $C(\hat{y}^{t_{\min}}, \hat{y}^{t_{\min}+1}, \dots, \hat{y}^{t_{\max}})$, where t_{\min} and t_{\max} are the first and last output time steps, **not counting the ignored outputs**.



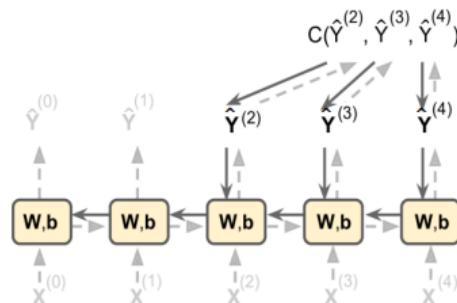
Backpropagation Through Time (2/3)

- ▶ 3. Propagate backward the gradients of that cost function through the unrolled network (represented by the solid arrows).
- ▶ 4. The model parameters are updated using the gradients computed during BPTT.

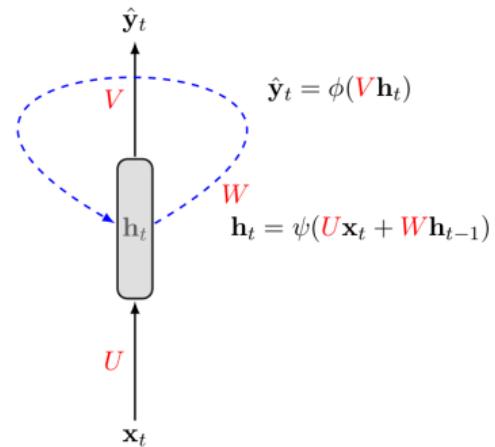


Backpropagation Through Time (3/3)

- ▶ The gradients **flow backward** through **all the outputs** used by the cost function, **not just through the final output**.
- ▶ For example, in the following figure:
 - The **cost function** is computed using the **last three outputs**, $\hat{y}^{(2)}$, $\hat{y}^{(3)}$, and $\hat{y}^{(4)}$.
 - Gradients flow through these three outputs, but **not through** $\hat{y}^{(0)}$ and $\hat{y}^{(1)}$.



BPTT Step by Step (1/20)

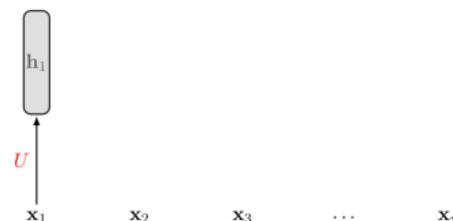




BPTT Step by Step (2/20)

x_1 x_2 x_3 \dots x_T

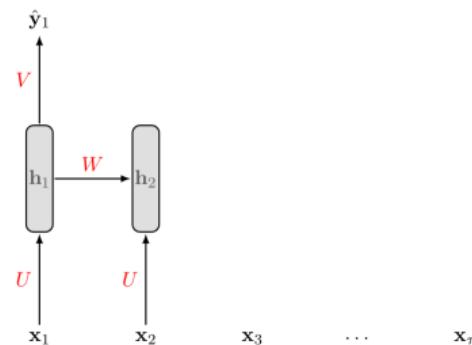
BPTT Step by Step (3/20)



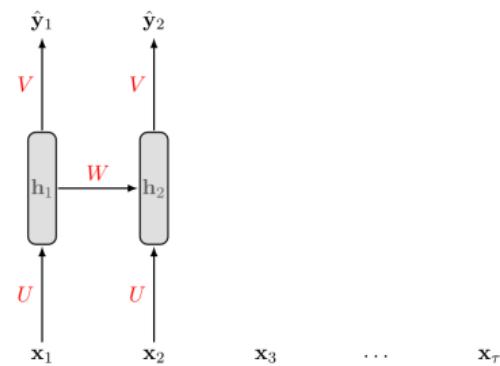
BPTT Step by Step (4/20)



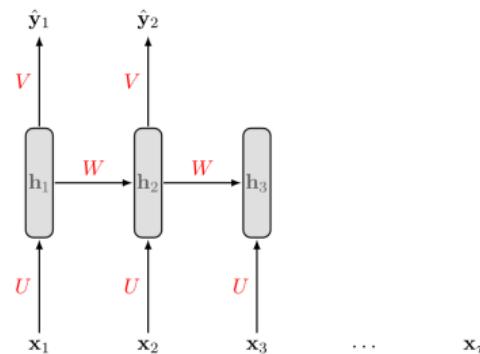
BPTT Step by Step (5/20)



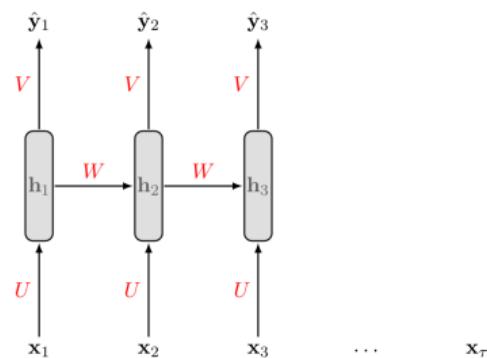
BPTT Step by Step (6/20)



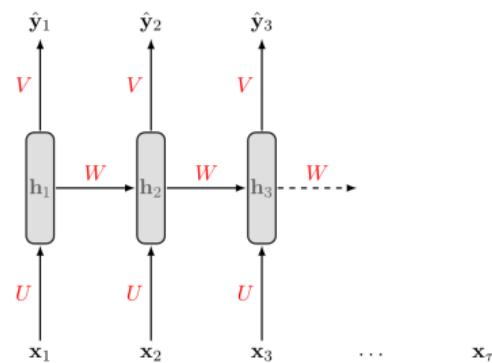
BPTT Step by Step (7/20)



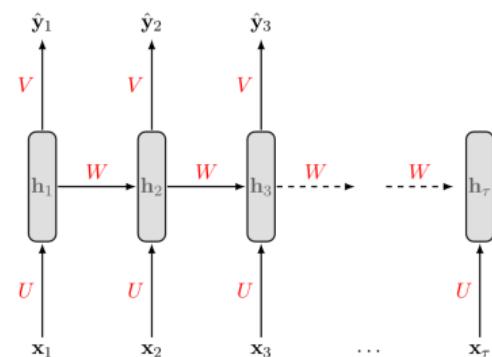
BPTT Step by Step (8/20)



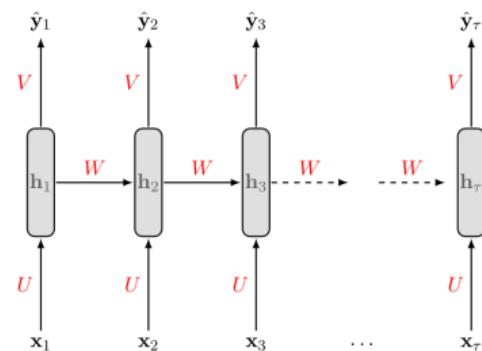
BPTT Step by Step (9/20)



BPTT Step by Step (10/20)



BPTT Step by Step (11/20)



BPTT Step by Step (12/20)

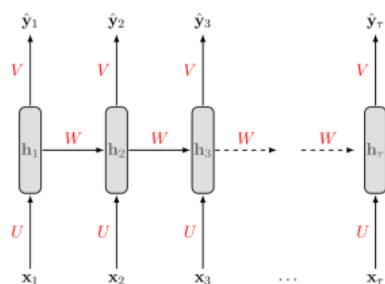
$$\mathbf{s}^{(t)} = \mathbf{u}^T \mathbf{x}^{(t)} + \mathbf{w} \mathbf{h}^{(t-1)}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{s}^{(t)})$$

$$\mathbf{z}^{(t)} = \mathbf{v} \mathbf{h}^{(t)}$$

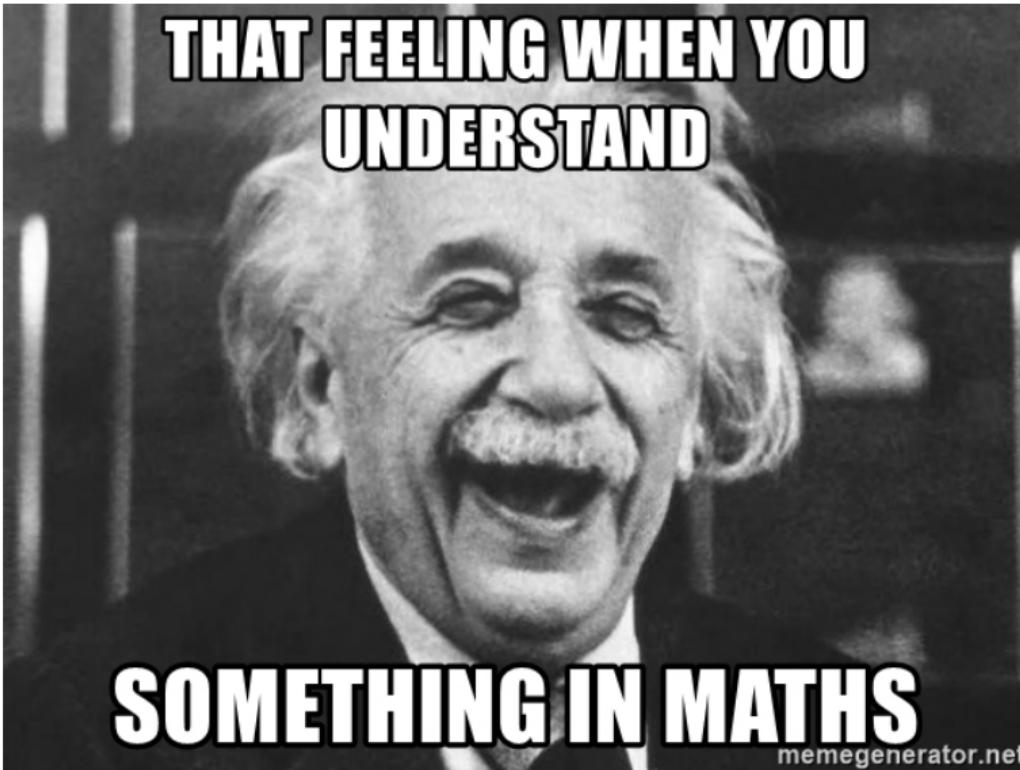
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{z}^{(t)})$$

$$J^{(t)} = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = - \sum y^{(t)} \log \hat{y}^{(t)}$$





**THAT FEELING WHEN YOU
UNDERSTAND**



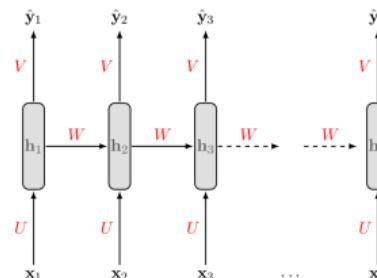
SOMETHING IN MATHS

memegenerator.net

BPTT Step by Step (13/20)

$$J^{(t)} = \text{cross_entropy}(y^{(t)}, \hat{y}^{(t)}) = - \sum y^{(t)} \log \hat{y}^{(t)}$$

- We treat the **full sequence** as one training example.
- The **total error E** is just the **sum of the errors at each time step**.
- E.g., $E = J^{(1)} + J^{(2)} + \dots + J^{(t)}$



BPTT Step by Step (14/20)

- ▶ $J^{(t)}$ is the total cost, so we can say that a 1-unit increase in v , w or u will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- ▶ The gradient is equal to the sum of the respective gradients at each time step t .
- ▶ For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial E}{\partial w} = \sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

$$\frac{\partial E}{\partial u} = \sum_t \frac{\partial J^{(t)}}{\partial u} = \frac{\partial J^{(3)}}{\partial u} + \frac{\partial J^{(2)}}{\partial u} + \frac{\partial J^{(1)}}{\partial u}$$

BPTT Step by Step (15/20)

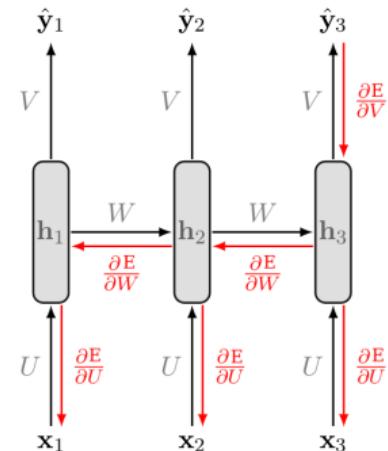
- ▶ Let's start with $\frac{\partial E}{\partial v}$.
- ▶ A change in v will only impact $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial J^{(3)}}{\partial v} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial v}$$

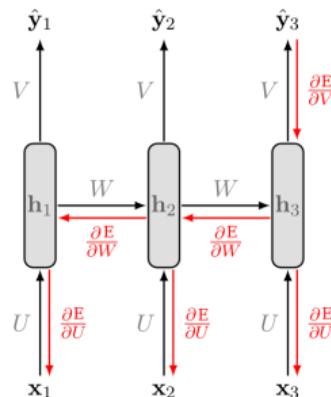
$$\frac{\partial J^{(2)}}{\partial v} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial v}$$

$$\frac{\partial J^{(1)}}{\partial v} = \frac{\partial J^{(1)}}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial v}$$



BPTT Step by Step (16/20)

- ▶ Let's compute the derivatives of $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial u}$, which are **computed the same**.
- ▶ A change in w at $t = 3$ will impact our cost J in 3 separate ways:
 1. When computing the value of $h^{(1)}$.
 2. When computing the value of $h^{(2)}$, which depends on $h^{(1)}$.
 3. When computing the value of $h^{(3)}$, which depends on $h^{(2)}$, which depends on $h^{(1)}$.

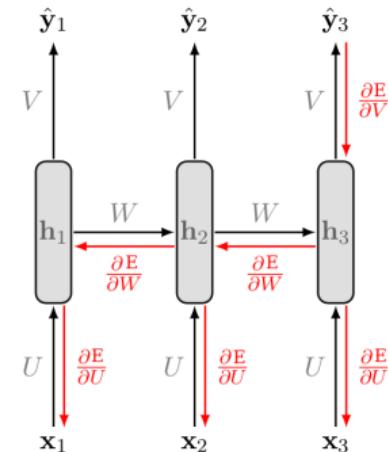


BPTT Step by Step (17/20)

- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

$$\frac{\partial J^{(1)}}{\partial w} = \frac{\partial J^{(1)}}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w}$$

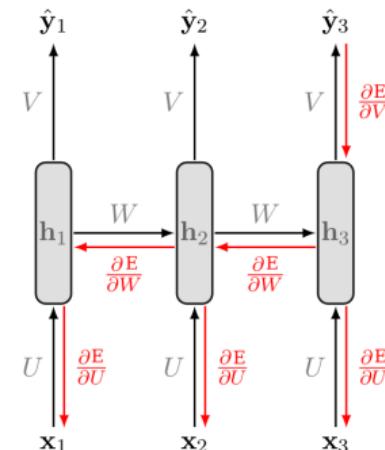


BPTT Step by Step (18/20)

- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

$$\begin{aligned} \frac{\partial J^{(2)}}{\partial w} = & \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial w} + \\ & \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w} \end{aligned}$$

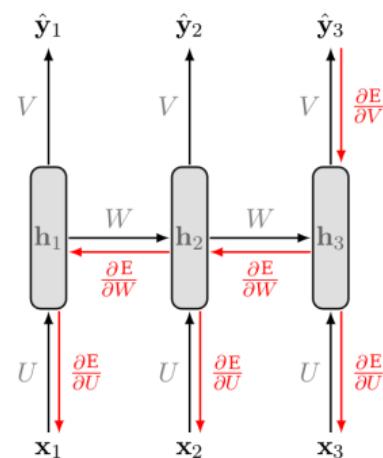


BPTT Step by Step (19/20)

- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

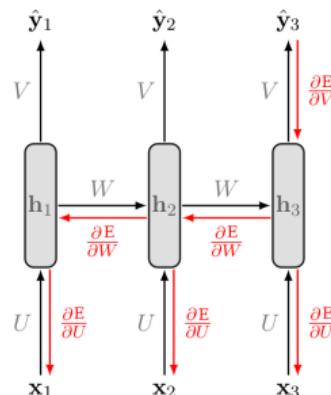
$$\begin{aligned} \frac{\partial J^{(3)}}{\partial w} &= \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial w} + \\ &\quad \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial w} + \\ &\quad \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w} \end{aligned}$$



BPTT Step by Step (20/20)

- More generally, a change in w will impact our cost $J^{(t)}$ on t separate occasions.

$$\frac{\partial J^{(t)}}{\partial w} = \sum_{k=1}^t \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial z^{(t)}} \frac{\partial z^{(t)}}{\partial h^{(t)}} \left(\prod_{j=k+1}^t \frac{\partial h^{(j)}}{\partial s^{(j)}} \frac{\partial s^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(k)}}{\partial s^{(k)}} \frac{\partial s^{(k)}}{\partial w}$$





RNN Problems

- ▶ Sometimes we only need to look at **recent information** to perform the present task.
 - E.g., **predicting the next word** based on the previous ones.
- ▶ In such cases, where the **gap between the relevant information and the place that it's needed** is **small**, RNNs can learn to use the past information.
- ▶ But, as that **gap grows**, RNNs become **unable to learn** to connect the information.
- ▶ RNNs may suffer from the **vanishing/exploding gradients problem**.



RNN References

- ▶ Ian Goodfellow et al., Deep Learning (Ch. 10)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 15)
- ▶ Understanding LSTM Networks
<http://colah.github.io/posts/2015-08-Understanding-LSTMs>
- ▶ CS224d: Deep Learning for Natural Language Processing
<http://cs224d.stanford.edu>



From RNNs to Transformers

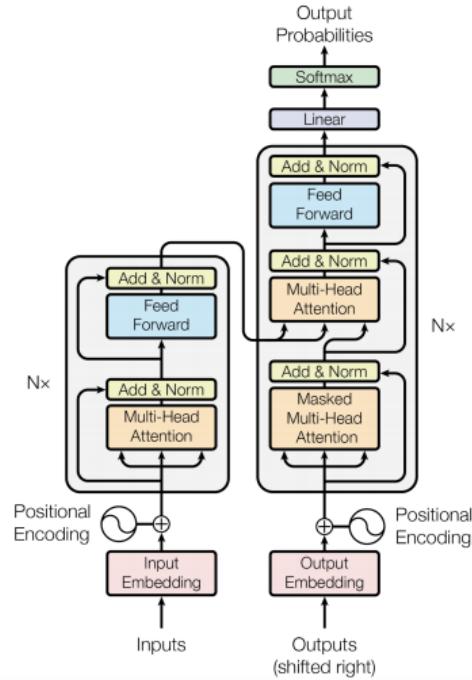


Problems with RNNs - Motivation for Transformers

- ▶ Sequential computations **prevents parallelization**
- ▶ Despite GRUs and LSTMs, RNNs still need attention mechanisms to deal with **long range dependencies**
- ▶ Attention gives us access to any state... Maybe we don't need the costly recursion?
- ▶ Then NLP can have deep models, solves our computer vision envy!

Attention is all you need! [Vaswani, 2017]

- ▶ Sequence-to-sequence model for Machine Translation
- ▶ Encoder-decoder architecture
- ▶ Multi-headed **self-attention**
 - Models context and no locality bias



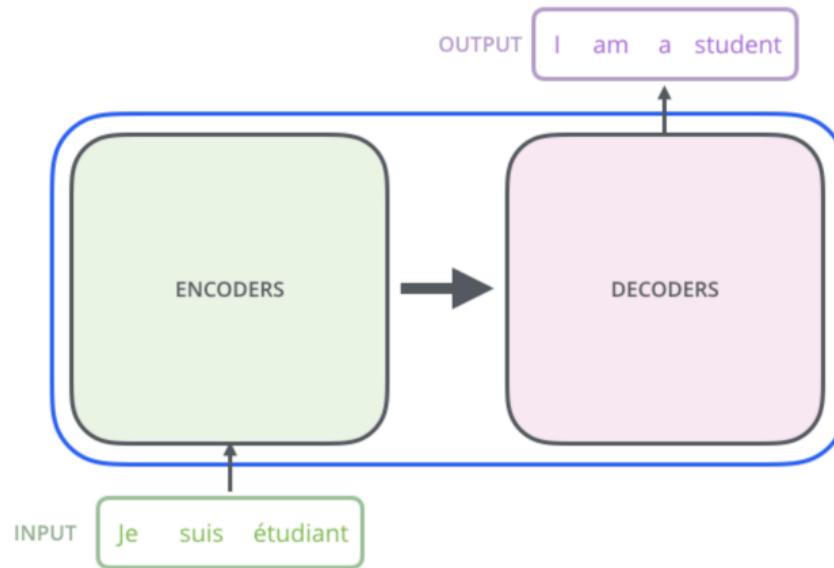
[Vaswani et al., 2017]



Transformers Step-by-Step



Understanding the Transformer: Step-by-Step

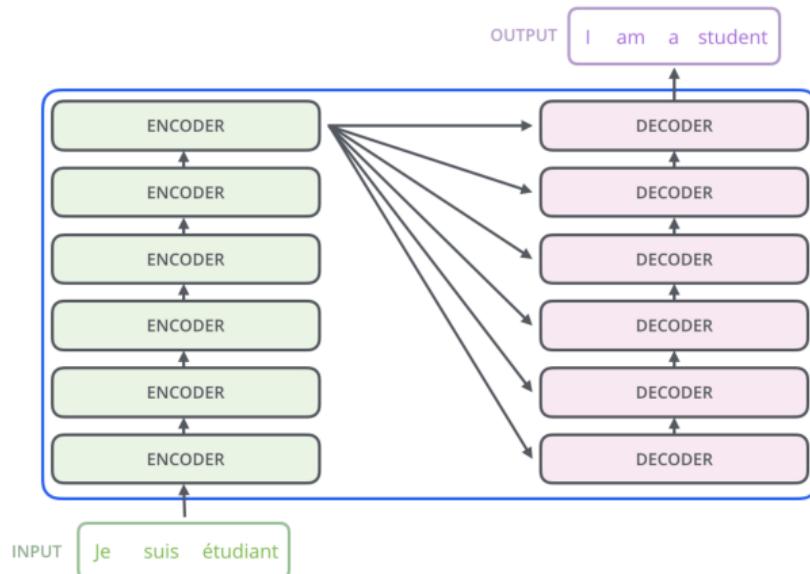


[Alammar, 2018]

Understanding the Transformer: Step-by-Step

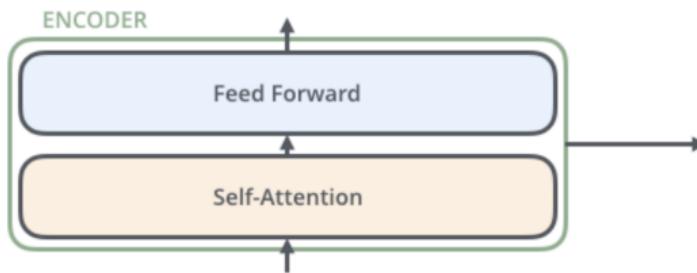
No recursion, instead
stacking encoder and
decoder blocks

- ▶ Originally: 6 layers
- ▶ BERT base: 12 layers
- ▶ BERT large: 24 layers
- ▶ GPT2-XL: 48 layers
- ▶ GPT3: 96 layers



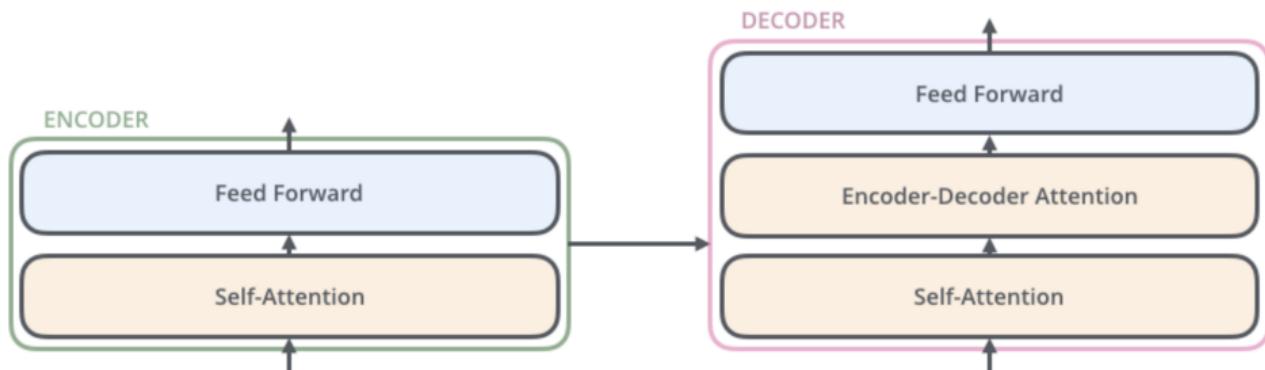
[Alammar, 2018]

The Encoder and Decoder Blocks



[Alammar, 2018]

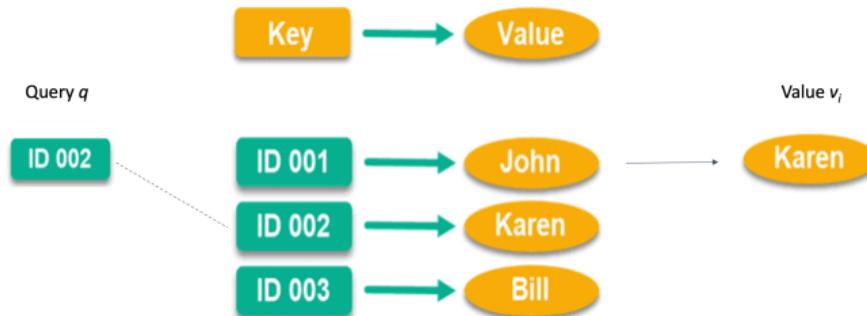
The Encoder Block



[Alammar, 2018]

Attention Preliminaries

Mimics the retrieval of a value v_i for a query q based on a key k_i in a database, but in a probabilistic fashion





Dot-Product Attention

- ▶ Queries, keys and values are vectors
- ▶ Output is a **weighted sum** of the values
- ▶ Weights are computed as the **scaled dot-product** (similarity) between the query and the keys

$$\text{Attention}(q, K, V) = \sum_i \text{Similarity}(q, k_i) \cdot v_i = \sum_i \frac{e^{q \cdot k_i / \sqrt{d_k}}}{\sum_j e^{q \cdot k_j / \sqrt{d_k}}} v_i$$

Output is a
row-vector

- ▶ Can stack multiple queries into a matrix Q

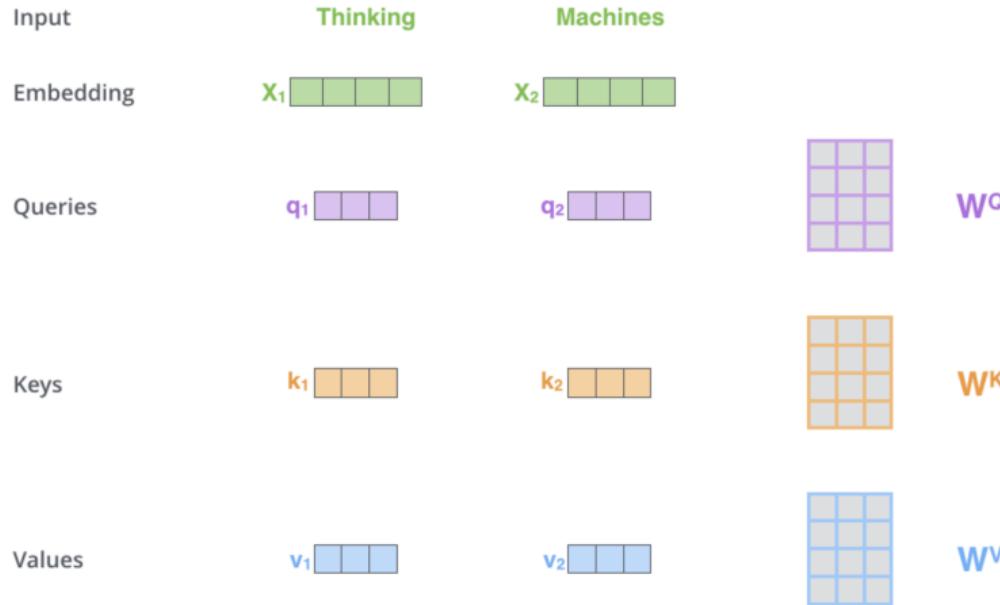
$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} \right) V$$

Output is again
a matrix

- ▶ Self-attention: Let the word embeddings be the queries, keys and values, i.e. **let the words select each other**

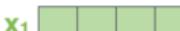
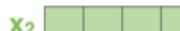
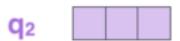


Self-Attention Mechanism



[Alammar, 2018]

Self-Attention Mechanism

Input	Thinking	Machines
Embedding	x_1 	x_2 
Queries	q_1 	q_2 
Keys	k_1 	k_2 
Values	v_1 	v_2 
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)	14	12
Softmax	0.88	0.12

[Alammar, 2018]

Self-Attention Mechanism in Matrix Notation

$$\mathbf{X} \times \mathbf{W}^Q = \mathbf{Q}$$

A diagram showing the multiplication of a green input matrix \mathbf{X} (3x3) by a purple weight matrix \mathbf{W}^Q (3x3) to produce a purple output matrix \mathbf{Q} (3x3).

$$\mathbf{X} \times \mathbf{W}^K = \mathbf{K}$$

A diagram showing the multiplication of a green input matrix \mathbf{X} (3x3) by an orange weight matrix \mathbf{W}^K (3x3) to produce an orange output matrix \mathbf{K} (3x3).

$$\mathbf{X} \times \mathbf{W}^V = \mathbf{V}$$

A diagram showing the multiplication of a green input matrix \mathbf{X} (3x3) by a light blue weight matrix \mathbf{W}^V (3x3) to produce a light blue output matrix \mathbf{V} (3x3).

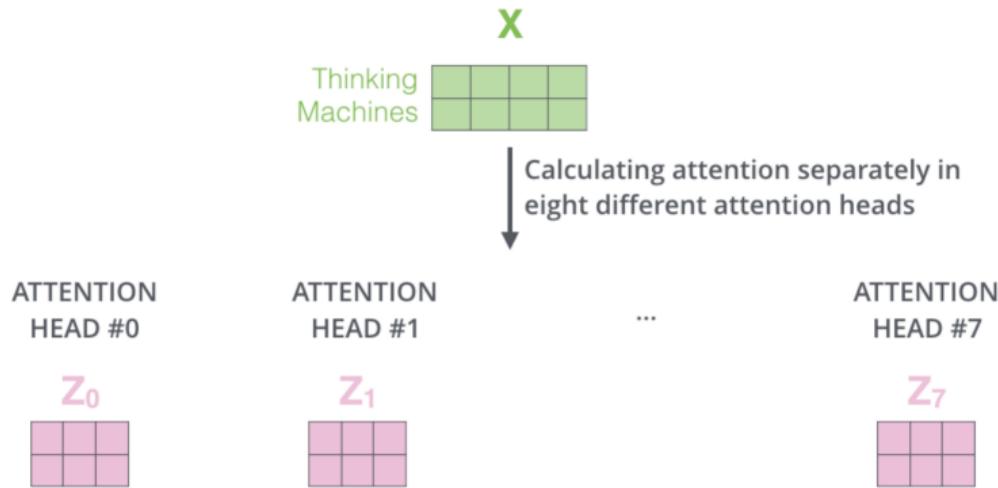
$$\text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} = \mathbf{Z}$$

The final computation of the self-attention mechanism. It shows the multiplication of the query matrix \mathbf{Q} (3x3) by the transpose of the key matrix \mathbf{K}^T (3x3), scaled by $\sqrt{d_k}$, followed by the multiplication with the value matrix \mathbf{V} (3x3) to produce the output matrix \mathbf{Z} (3x3).

[Alammar, 2018]

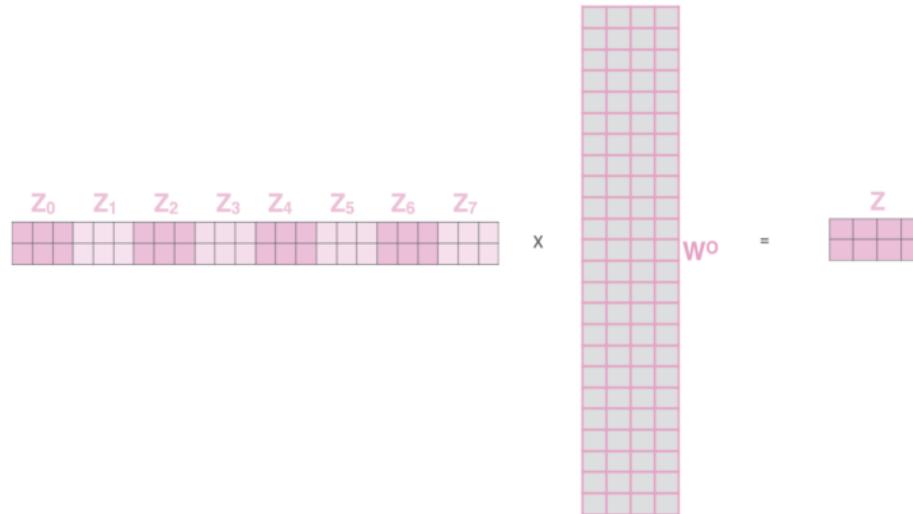


Multi-Headed Self-Attention



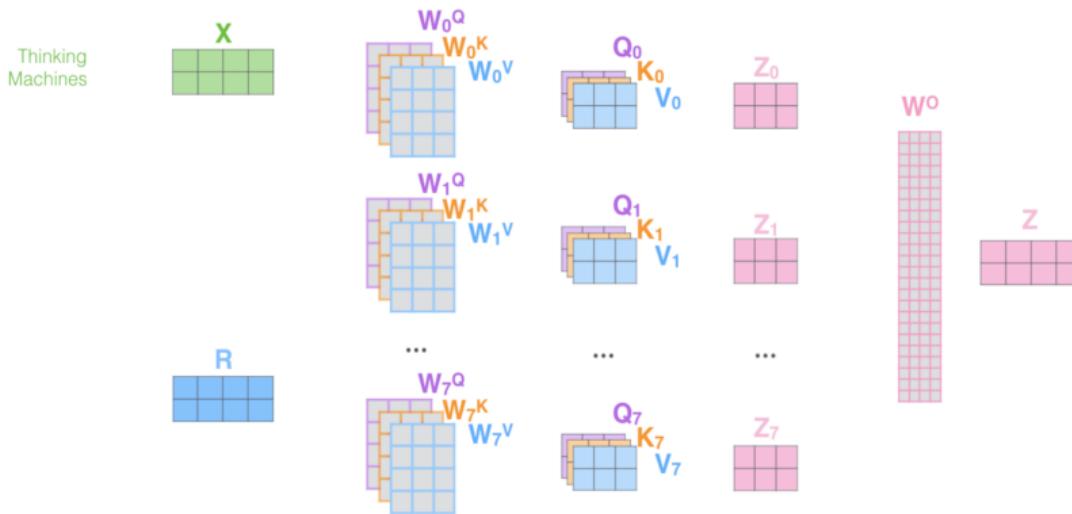
[Alammar, 2018]

Multi-Headed Self-Attention



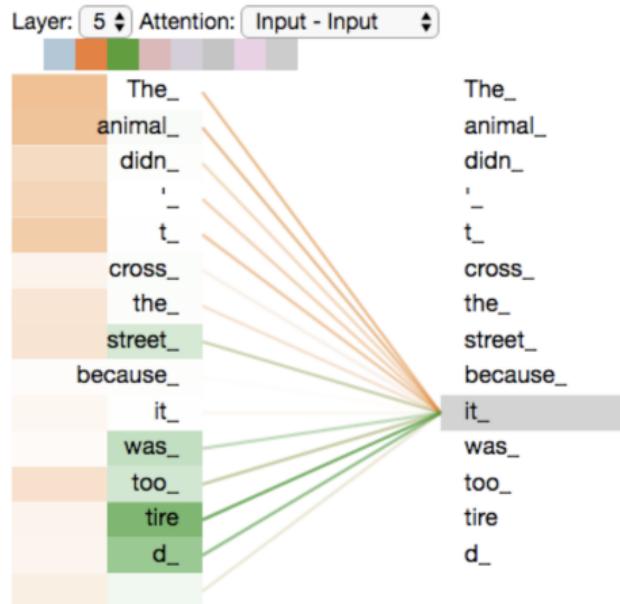
[Alammar, 2018]

Self-Attention: Putting It All Together



[Alammar, 2018]

Attention Visualized

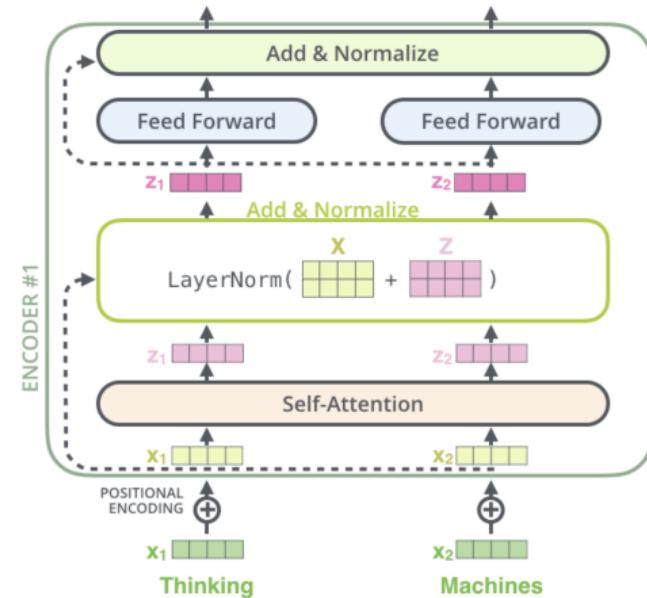


[Alammar, 2018]

The Full Encoder Block

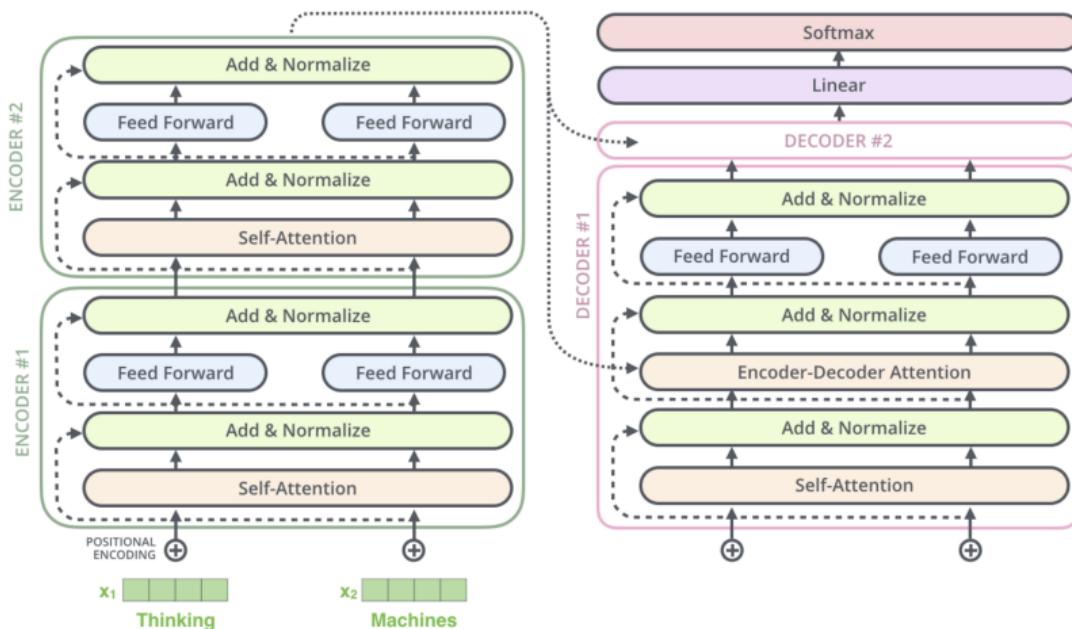
Encoder block consisting of:

- ▶ Multi-headed self-attention
- ▶ Feedforward NN (FC 2 layers)
- ▶ Skip connections
- ▶ Layer normalization - Similar to batch normalization but computed over features (words/tokens) for a single sample



[Alammar, 2018]

Encoder-Decoder Architecture - Small Example



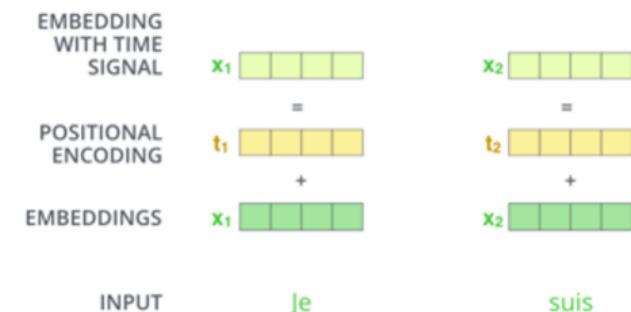
[Alammar, 2018]

Positional Encodings

Encoder block consisting of:

- ▶ Attention mechanism has no locality bias - **no notion of word order**
- ▶ **Add positional encodings** to input embeddings to let model learn relative positioning

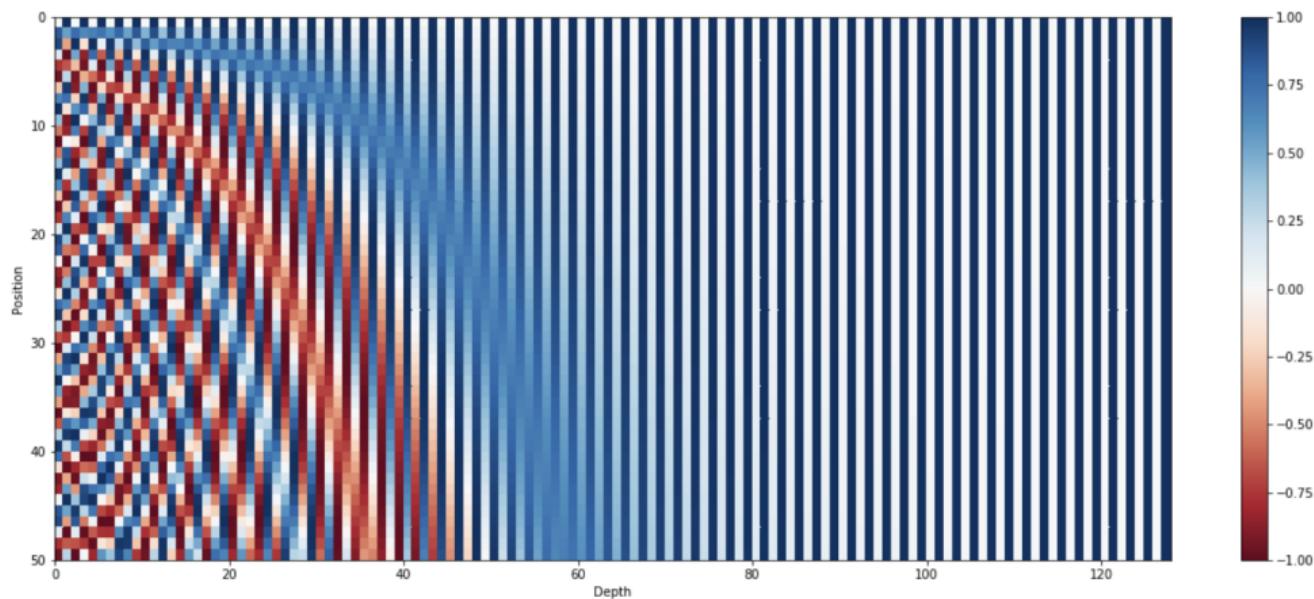
$$\text{PE}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right)$$



$$\text{PE}(\text{pos}, 2i + 1) = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right)$$

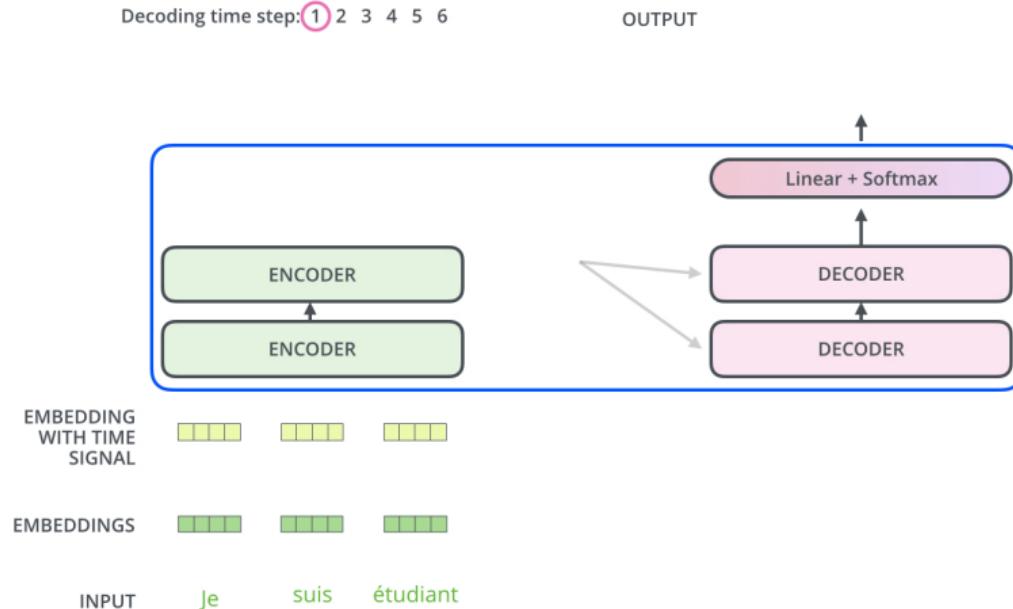
[Alammar, 2018]

Positional Encodings



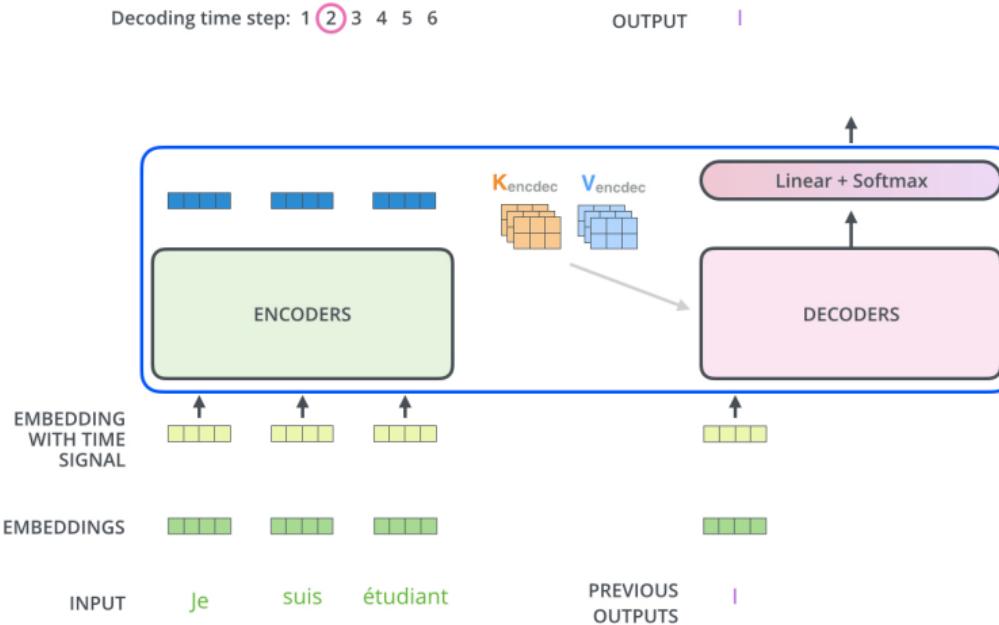
[Kazemnejad, 2019]

Let's start the encoding!



[Alammar, 2018]

Decoding procedure

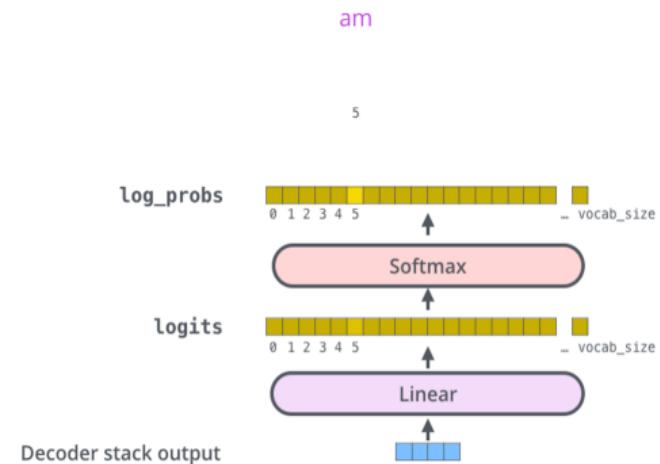


[Alammar, 2018]

Producing the output text

Encoder block consisting of:

- ▶ The output from the decoder is passed through a final fully connected **linear layer** with a **softmax** activation function
- ▶ Produces a probability distribution over the pre-defined vocabulary of output words (tokens)
- ▶ **Greedy decoding** picks the word with the highest probability at each time step



[Alammar, 2018]

Training Objective

Target Model Outputs

Output Vocabulary: a am I thanks student <eos>

position #1	0.0	0.0	1.0	0.0	0.0	0.0
-------------	-----	-----	------------	-----	-----	-----

position #2	0.0	1.0	0.0	0.0	0.0	0.0
-------------	-----	------------	-----	-----	-----	-----

position #3	1.0	0.0	0.0	0.0	0.0	0.0
-------------	------------	-----	-----	-----	-----	-----

position #4	0.0	0.0	0.0	0.0	1.0	0.0
-------------	-----	-----	-----	-----	------------	-----

position #5	0.0	0.0	0.0	0.0	0.0	1.0
-------------	-----	-----	-----	-----	-----	------------

a am I thanks student <eos>

Trained Model Outputs

Output Vocabulary: a am I thanks student <eos>

position #1	0.01	0.02	0.93	0.01	0.03	0.01
-------------	------	------	-------------	------	------	------

position #2	0.01	0.8	0.1	0.05	0.01	0.03
-------------	------	------------	-----	------	------	------

position #3	0.99	0.001	0.001	0.001	0.002	0.001
-------------	-------------	-------	-------	-------	-------	-------

position #4	0.001	0.002	0.001	0.02	0.94	0.01
-------------	-------	-------	-------	------	-------------	------

position #5	0.01	0.01	0.001	0.001	0.001	0.98
-------------	------	------	-------	-------	-------	-------------

a am I thanks student <eos>



[Alammar, 2018]

Complexity Comparison

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$

[Vaswani et al., 2017]

Results

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1		$3.3 \cdot 10^{18}$
Transformer (big)	28.4	41.0		$2.3 \cdot 10^{19}$

[Vaswani et al., 2017]



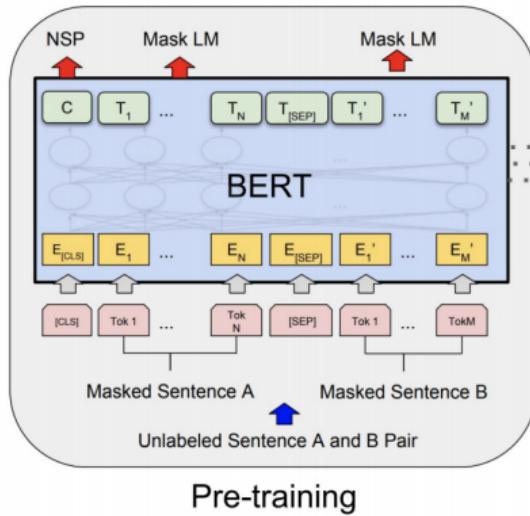
BERT

Bidirectional Encoder Representations from Transformers

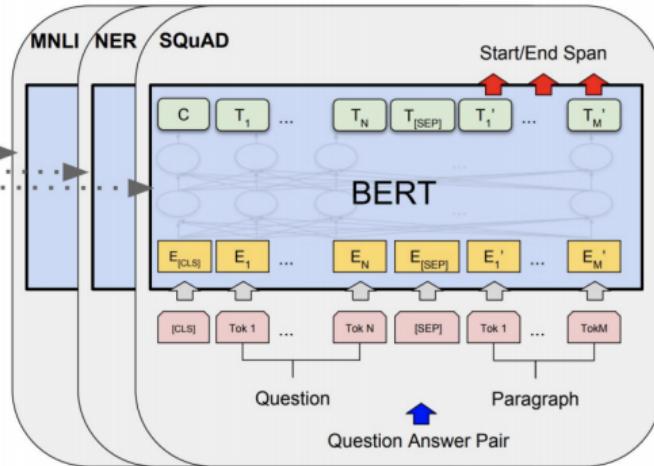
- ▶ Self-supervised pre-training of Transformers encoder for language understanding
- ▶ Fine-tuning for specific downstream task



BERT Training Procedure



Pre-training



Fine-Tuning

[Devlin et al., 2018]



BERT Training Objectives

Masked Language Modelling

the man went to the [MASK] to buy a [MASK] of milk

↑ ↑
store gallon

Next Sentence prediction

Sentence A = The man went to the store.

Sentence B = He bought a gallon of milk.

Label = IsNextSentence

Sentence A = The man went to the store.

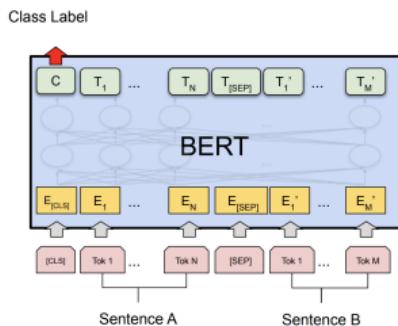
Sentence B = Penguins are flightless.

Label = NotNextSentence

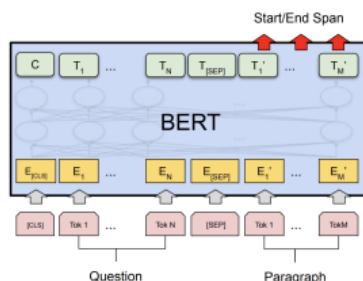
[Devlin et al., 2018]

BERT Fine-Tuning Examples

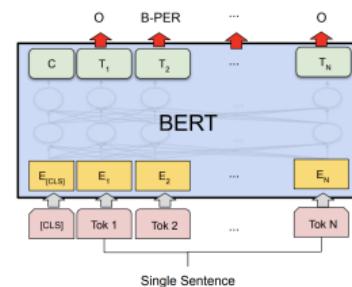
Sentence Classification



Question Answering



Named Entity Recognition



[Devlin et al., 2018]



How good are transformers?

- ▶ Scaling up **models size** and amount of **training data** helps a lot
- ▶ Best model is 10B (!!!) parameters
- ▶ Two models have already surpassed human performance!!!
- ▶ Exact **pre-training objective** (MLM, NSP, corruption) doesn't matter too much
- ▶ SuperGLUE benchmark:

Rank	Name	Model	URL	Score	BoolQ	CB	COPA	MultIRC	ReCoRD	RTE	WiC	WSC	AX-g	AX-b	
1	ERNIE Team - Baidu	ERNIE 3.0		90.6	91.0	98.6/99.2	97.4	88.6/63.2	94.7/94.2	92.6	77.4	97.3	92.7/94.7	68.6	
+	2	Zirui Wang	T5 + UDG, Single Model (Google Brain)		90.4	91.4	95.8/97.6	98.0	88.3/63.0	94.2/93.5	93.0	77.9	96.6	92.7/91.9	69.1
+	3	DeBERTa Team - Microsoft	DeBERTa / TuringNLv4		90.3	90.4	95.7/97.6	98.4	88.2/63.7	94.5/94.1	93.2	77.5	95.9	93.3/93.8	66.7
	4	SuperGLUE Human Baselines	SuperGLUE Human Baselines		89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	99.3/99.7	76.6
+	5	T5 Team - Google	T5		89.3	91.2	93.9/96.8	94.8	88.1/63.3	94.1/93.4	92.5	76.9	93.8	92.7/91.9	65.6
+	6	Huawei Noah's Ark Lab	NEZHA-Plus		86.7	87.8	94.4/96.0	93.6	84.6/55.1	90.1/89.6	89.1	74.6	93.2	87.1/74.4	58.0

[Raffel et al., 2019]



Practical Examples



BERT in low-latency production settings

GOOGLE \ TECH \ ARTIFICIAL INTELLIGENCE

Google is improving 10 percent of searches by understanding language context

Say hello to BERT

By Dieter Bohn | @backlon | Oct 25, 2019, 3:01am EDT

Bing says it has been applying BERT since April

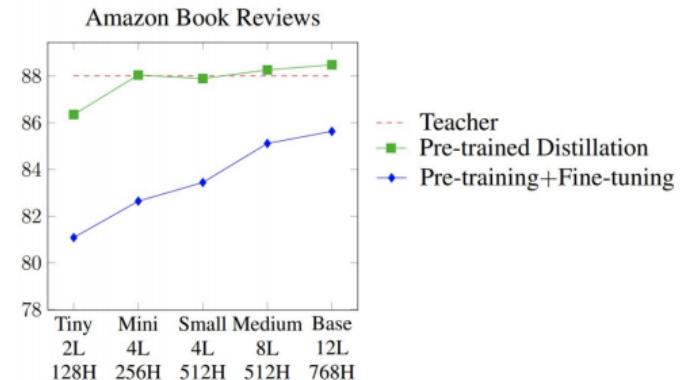
The natural language processing capabilities are now applied to all Bing queries globally.

[George Nguyen](#) on November 19, 2019 at 1:38 pm

[Devlin, 2020]

Distillation

- ▶ Modern pre-trained language models are **huge** and very **computationally expensive**
- ▶ How are these companies applying them to low-latency applications?
- ▶ Distillation!
 - Train SOTA **teacher model** (pre-training + fine-tuning)
 - Train smaller **student model** that **mimics** the teacher's output on a large dataset on unlabeled data
- ▶ Distillation works *much* better than pre-training + fine-tuning with smaller model



[Devlin, 2020] [Turc, 2020]



Transformers in TensorFlow using HuggingFace 😊

- ▶ The [HuggingFace Library](#) contains a majority of the recent pre-trained State-of-the-art NLP models, as well as over 4 000 community uploaded models
- ▶ Works with both [TensorFlow](#) and [PyTorch](#)



HUGGING FACE

[Back to home](#)

All Models and checkpoints

Also check out our list of [Community contributors](#) 🎉 and [Organizations](#) 🌐.

Search models... Tags: All ▾ Sort: Most downloads ▾

- bert-base-uncased ★
- deepset/bert-large-uncased-whole-word-masking-squad2
- distilbert-base-uncased ★
- dccuchile/bert-base-spanish-wwm-cased ★
- microsoft/xprophetnet-large-wiki100-cased-xglue-ntg ★
- deepset/roberta-base-squad2 ★
- jplu/tf-xlm-roberta-base ★
- cl-tohoku/bert-base-japanese-whole-word-masking
- distilroberta-base ★
- bert-base-cased ★
- xlm-roberta-base ★



Transformers in TensorFlow using HuggingFace 😊

```
from transformers import BertTokenizerFast, TFBertForSequenceClassification
from datasets import load_dataset
import tensorflow as tf

dataset = load_dataset("imdb").shuffle()
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

train_encodings = tokenizer(dataset['train']['text'], truncation=True, padding=True)
train_dataset = tf.data.Dataset.from_tensor_slices((dict(train_encodings), dataset['train']['label']))
val_dataset = ... // Analogously

optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss=model.compute_loss)
model.fit(train_dataset.batch(16), epochs=3, batch_size=16)

model.evaluate(val_dataset.batch(16), verbose=0)
```



Transformers in TensorFlow using HuggingFace 😊

```
from transformers import BertTokenizerFast, TFBertForSequenceClassification
from datasets import load_dataset
import tensorflow as tf

dataset = load_dataset("imdb").shuffle()
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

train_encodings = tokenizer(dataset['train']['text'], truncation=True, padding=True)
train_dataset = tf.data.Dataset.from_tensor_slices((dict(train_encodings), dataset['train']['label']))
val_dataset = ... // Analogously

optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss=model.compute_loss)
model.fit(train_dataset.batch(16), epochs=3, batch_size=16)

model.evaluate(val_dataset.batch(16), verbose=0)
```



Transformers in TensorFlow using HuggingFace 😊

```
from transformers import BertTokenizerFast, TFBertForSequenceClassification
from datasets import load_dataset
import tensorflow as tf

dataset = load_dataset("imdb").shuffle()
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

train_encodings = tokenizer(dataset['train']['text'], truncation=True, padding=True)
train_dataset = tf.data.Dataset.from_tensor_slices((dict(train_encodings), dataset['train']['label']))
val_dataset = ... // Analogously

optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss=model.compute_loss)
model.fit(train_dataset.batch(16), epochs=3, batch_size=16)

model.evaluate(val_dataset.batch(16), verbose=0)
```



Transformers in TensorFlow using HuggingFace 😊

```
from transformers import BertTokenizerFast, TFBertForSequenceClassification
from datasets import load_dataset
import tensorflow as tf

dataset = load_dataset("imdb").shuffle()
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

train_encodings = tokenizer(dataset['train']['text'], truncation=True, padding=True)
train_dataset = tf.data.Dataset.from_tensor_slices((dict(train_encodings), dataset['train']['label']))
val_dataset = ... // Analogously

optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss=model.compute_loss)
model.fit(train_dataset.batch(16), epochs=3, batch_size=16)

model.evaluate(val_dataset.batch(16), verbose=0)
```



Wrap Up

Summary

- ▶ Transformers have blown other architectures out of the water for NLP
- ▶ Get rid of recurrence and rely on **self-attention**
- ▶ NLP pre-training using **Masked Language Modelling**
- ▶ Most recent improvements using **larger models** and **more data**
- ▶ **Distillation** can make model serving and inference more tractable

